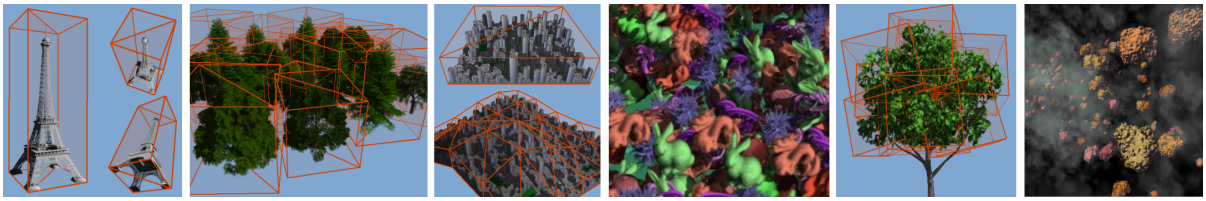
Preprint. The definitive version is available at www.blackwell-synergy.com

COMPUTER GRAPHICS *forum*
Volume 28 (2009)

Volumetric Billboards

Philippe Decaudin[1,2] and Fabrice Neyret[1]

[1]LJK/IMAG-INRIA Grenoble, France
[2]CASIA Beijing, China



Abstract

We introduce an image-based representation, called volumetric billboards, *allowing for the real-time rendering of semi-transparent and visually complex objects arbitrarily distributed in a 3D scene. The representation offers full parallax effect from any viewing direction and improved anti-aliasing of distant objects. It correctly handles transparency between multiple and possibly overlapping objects without requiring any primitive sorting. Furthermore, volumetric billboards can be easily integrated into common rasterization-based renderers, which allows for their concurrent use with polygonal models and standard rendering techniques such as shadow-mapping.*
The representation is based on volumetric images of the objects and on a dedicated real-time volume rendering algorithm that takes advantage of the GPU geometry shader. Our examples demonstrate the applicability of the method in many cases including levels-of-detail representation for multiple intersecting complex objects, volumetric textures, animated objects and construction of high-resolution objects by assembling instances of low-resolution volumetric billboards.

Keywords: *image-based rendering, billboards, real-time rendering, geometry shader*

1. Introduction

Image-based representations are attractive alternatives to classical polygonal models for real-time 3D applications. They are able to represent geometrically complex objects independently of their complexity. This includes objects made of many small, thin and possibly disconnected elements that become individually indistinguishable at a distance but still predominantly contribute to the overall shape of the object. Examples of such objects are commonly found in nature (plants, trees, objects covered by fuzzy material,...) and in man-made detailed structures (complex constructions, metallic structures,...). In mesh representations, these objects are difficult to visualize in real-time because of the large number of primitives required to render them. Moreover, they are also prone to aliasing artifacts since the primitives can be much smaller than the size of a pixel on screen. These issues become all the more acute when the object is viewed at a distance. Geometric levels-of-detail techniques such as polygon reduction do not help in such cases since thin and possibly disconnected elements cannot be merged into larger polygons without deteriorating the visual appearance of the model, even when seen from far away. Indeed, to be accurately filtered according to the viewing distance, a set of opaque elements that do not entirely fill a space region should be replaced with a semi-transparent primitive.

Image-based techniques are well suited to represent semi-transparency, this is done by storing an opacity value in the images. Furthermore, they ease the representation of anti-aliased levels-of-detail since the images can be progressively filtered and stored as MIP-maps. However, image-based representations used in real-time applications often suffer from unpleasant visual artifacts, such as limited parallax effects and popping. Moreover, the integration of multiple and arbitrarily placed objects in a 3D scene presents the problem

of correctly handling transparency. The semi-transparent regions of different images must be composited in the right order to yield a correct result. For non-overlapping objects, the immediate solution would be to sort their bounding boxes (or bounding polyhedra in the general case) and to render them in the appropriate order. But sorting them accurately with respect to the current view projection is non-trivial [CMSW04], and not always possible (see Figure 1). For overlapping objects, this implies that the rendering algorithm cannot process the objects one at a time and thus should handle multiple objects altogether in order to intermix their data.

Figure 1: *Boxes cannot always be sorted in a back-to-front order wrt. the viewer.*

In this paper, we introduce an image-based representation, called *volumetric billboards*, which efficiently solves these issues. In particular, this representation is able to properly render a set of semi-transparent objects that may overlap with each other or intersect other objects represented by polygonal models. The basic idea is to directly rely on volumetric images of the objects possibly generated from the voxelization of their polygonal models. The volumetric billboards are defined as a set of 3D cells arbitrarily placed in a 3D scene and filled with the volumetric images. The volumetric images are MIP-mapped and stored into 3D textures. The cell vertices have 3D texture coordinates that map a volumetric image to the cell. They are rendered with a dedicated slice-based volume rendering algorithm that renders all the volumetric billboards at once and correctly handles transparency without requiring any sorting. The rendering algorithm also adapts to render distant volumetric billboards more efficiently, while ensuring accurate anti-aliasing and MIP-map filtering.

As our results show, volumetric billboards allow for the real-time visualization of many complex objects, with full parallax from any viewing direction and anti-aliased levels-of-detail. These objects can be seamlessly integrated into a scene containing polygon-based objects. Furthermore, volumetric billboards can be mapped onto a surface to cover it like a volumetric texture (in the sense of [KK89]).

2. Previous Work

Our approach is mainly related to image-based rendering and real-time volume rendering techniques. A large literature has been devoted to these two fields, so we will focus only on techniques closely related to ours. Extensive and recent reviews can be found in [JWP05] for image-based rendering, and in [HKRS*06] for real-time volume rendering.

The billboard technique and its extensions, such as [DDSD03], represent an object by one or more intersecting textured quads. They share the advantages of image-based representation, such as low rendering cost and MIP-map filtering with the distance. However, the correct handling of semi-transparent billboard regions has not been addressed satisfactorily. Even if the original object is opaque, its contours will become semi-transparent with MIP-map filtering. Sorting billboards for correct transparency is costly and requires splitting the (numerous) billboards which intersect each other. Furthermore, if several such intersecting objects have to be considered, the sorting and splitting must be done globally (at runtime if objects move).

Other impostor-based techniques use a set of cached images of different views to represent an object [MS95, TK96]. To better capture and render a 3D shape, the images can be layered [Sch98], pre-warped [OBM00], or augmented with depth information *e.g.*, [PO06, ABB*07]. However, those methods are restricted to essentially opaque objects, and cannot represent objects that become fuzzy or semi-transparent at a distance. Light fields [LH96, GGSC96] are an interesting approach to compress the aspect from all directions of any complex object (real or CG). But it does not permit the mixing with other 3D primitives in the same area. Moreover, the shading is frozen in the data, or 2 extra dimensions should be used to include this degree of freedom, rising the light field to 6D.

Real-time volume rendering has focused predominantly on texture-based approaches. Slice-based methods [CCF94, LL94] use a stack of parallel planes as a proxy geometry to sample the volume and evaluate the volume rendering integral by blending the slices. These methods are still widely used in interactive volume visualization applications because of their easy implementation and their ability to benefit from hardware trilinear texture interpolation and pre-integration schemes to generate high quality images at a high framerate [EE02]. The recent development of programmable graphics hardware has also led to the emergence of GPU-based implementations of raycasting [KW03]. This approach eases optimizations such as early-ray termination and empty-space skipping. Even though, GPU specific issues have to be considered carefully to avoid performance drop [HSHH07]. Both slice-based and raycasting approaches have mainly targeted visualization of a single volume (such as scientific simulation results or medical data). Integrating multiple volumes in a 3D scene includes issues related to data intermixing of the multiple and possibly overlapping volumes, and integration with polygonal models. [CS99] discriminates three levels of data intermixing. The simplest processes the volumes separately and merges the final images, which obviously produces unrealistic effects. The intermediate level mixes the visual contribution of each volume present at a same location during voxel traversing. The more complex level intermixes the illumination models of overlapping voxels. These last two levels fit our requirements, but the intermediate one has the advantage of generality: Mixing illumination models in the general case is still an open problem which the intermediate level avoids by mixing voxels after their shading evaluation. The slight loss in

image quality is indeed acceptable for real-time rendering. This is the data intermixing level considered in this paper.

Regarding GPU-raycasting, the design of algorithms able to handle the data intermixing leads to complex and costly solutions. Basically, if one wants to render several semi-transparent volume objects (that may intersect one another), they have to be considered at once during the ray integration. This suggests a dedicated compositing step [Gri05] and organizing the volumes into a larger spatial data structure to be handled by the GPU.

On the other hand, slice-based rendering directly solves the problem of mixing volumes with polygonal objects present in the scene and pre-rasterized: It produces pixel fragments at their correct z-position that are then tested against the current value stored in the z-buffer before being composited with the fragments coming from the next slice. Regarding correct transparency handling of multiple semi-transparent volumes, a solution dedicated to the rendering of volumetric textures (in the sense of [KK89]) mapped onto a surface has been proposed by [LDS02]. Their scheme renders a set of prism-shaped cells filled with a semi-transparent 3D texture, without requiring them to be sorted. This interesting property leads us to build our solution on this slicing scheme. We generalize their scheme to account for important issues related to cells arbitrarily placed in a 3D scene. This includes the handling of disconnected cells possibly filled with different textures as well as the handling of MIP-mapped volume for distant objects. We also enhance the efficiency of the scheme with an improved prism/plane intersection algorithm implemented on GPU (as a geometry shader) to slice the cells.

3. The volumetric billboards model

3.1. Representation

Our representation is composed of a set of triangular prism-shaped 3D cells arbitrarily positioned in the scene with 3D texture coordinates at their vertices, and a set of volume data stored into 3D textures which are mapped into the cells. In typical uses, the cells are likely to overlap.

The cell: Triangular prism-shaped cells are very convenient: First, two prisms can form a box, which is the most common bounding polyhedron for an object (Figure 2-*left*). Second, the plane/cell intersection algorithm to slice the cells can be expressed in a compact and efficient manner for prisms (see Section 3.3.2). Finally, prisms can be generated by extruding any triangular mesh, which facilitates the creation of volumetric billboards for covering a surface, and thus extends their use to volumetric texturing. Our cell shape is a generalized triangular prism defined as an extruded triangle where the three extruded edges do not need to be parallel or to have the same length.

The volume: Unlike volumes generated from medical or simulation applications for which the data (*e.g.*, a density) is

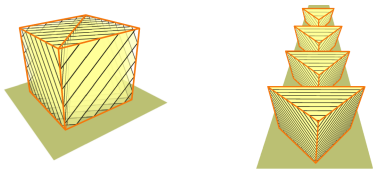


Figure 2: Left: *A cube made of two prism-shaped cells and sliced parallel to the screen.* Right: *The interval between slices is adapted with the distance.*

interpreted as a participating medium, our data are meant to directly represent *appearance*, as in other image-based representations. In particular, color and opacity are potentially view-dependent, and extra information such as normals might be available. This impacts the way volumes are built from source object description, how the volumes are filtered for MIP-mapping, and how they are rendered. Moreover, the source objects do not only consist of geometry, especially if they were originally designed for high-quality rendering. They can also have textures, precalculated data such as ambient occlusion [ZIK98], and even complex shaders. The fact that we build our volume from object appearance (using pre-renderings) allows us to capture all these features. However, our scope is the high performance rendering of the volumes. For efficiency in terms of storage and pixel shading, the implementation used for our examples do not encode view-dependent information in the 3D textures. Our voxels simply contain classical RGBA components and possibly a normal N. But the volume data and associated MIP-map pyramids are built so that the impact of this simplification is reduced, as described in the next section.

3.2. Volume data generation

Colors and transparency: Regarding color and transparency, the offline creation of volume data from source objects is inspired from [DN04]. In that paper dedicated to forest rendering, voxelization of trees is performed by rendering a stack of top-view slices of the trees. Each slice image is obtained by fitting an orthographic camera above the trees and setting its near and far clip planes to match the slice position and thickness (a similar approach is also used for opacity shadow maps in [KN01]). RGBA image components are then used to fill the corresponding slice of the volume. Thus, the volume encodes the top-view appearance of the trees. This choice was made because the primary goal of [DN04] is forest flyover. In our case, volumetric billboards are viewed from any direction. To produce volume data that remains acceptable from any view, we extend the algorithm to combine the source objects appearance rendered from the six axis directions. Six stacks of slices (one for each direction) are generated. Each voxel of the volume is then filled by a color and transparency resulting from the combination of the values sampled in each direction. A neutral combination function would be simple averaging. However, depending on the kind of content, it can be interesting to favor some features such as the opacity of thin objects like straws or leaves. In this case, a *max* function might be prefered to preserve them.

Thus, the function choice is a parameter of the voxelization process to be set by the user. This representation can be seen as a volume in which voxel RGBA anisotropy is sampled in six directions in the spirit of what [Nom95] does for leaves in his volumetric texture of trees.

Since the view-dependent information has been merged in the resulting volume, building a MIP-map pyramid by filtering this volume at lower resolutions would lead to poor quality MIP-maps which tend to become over-transparent. To alleviate this issue, we preserve the view-dependent information as long as possible during the process (Figure 3). First we separately build six volume MIP-map pyramids from the six stacks of slices rendered previously. Each MIP-map pyramid is built following the algorithm of [DN04] that takes into account the occlusion that occurs between the slices in the stack view direction: In that direction, instead of averaging the slices by pair, these pairs are *composited*. The composition uses the back-to-front alpha blending equation which thus accounts for occlusion. Finally for each MIP-map level, the six volumes of the six pyramids are combined into one using the combination function defined by the user.

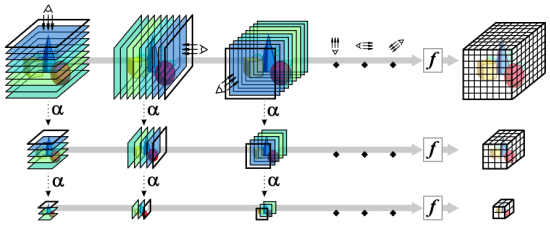


Figure 3: *Volume MIP-maps construction from 6 view directions (α : alpha blending operator in the view direction, f : user defined voxel combination function).*

Reflectance: The image-based representations commonly used in real-time applications either encode the normal for runtime shading (*e.g.*, bump-maps) or precompute the shading in the color texture (this is frequently done for 2D billboards, especially when they represent plants or trees). Moreover, nowadays a part of the lighting is often precomputed in textures even for quality-rendering applications (*e.g.*, ambient occlusion).

We combine both approaches: When processing the source objects we capture images of normals as well, and in the captured colors we can account for any feature the user prefers to precompute rather than evaluate at runtime in a pixel shader. However, sampling normals has limitations. *E.g.*, for objects made of thin elements (for instance, objects of Figures 8, 11 and 13), a single normal does not bring enough information to represent the filtered reflectance. In such cases, normals are not reliable and precalculation is preferable. Still, we might approximate the lighting as a separable low-frequency and high-frequency lighting, precalculating the local high-frequency lighting and storing a low-frequency normal for runtime shading.

To MIP-map our captured normals, we rely on the normal filter proposed in [Tok05]. Although approximate, this works well in practice for objects with a fairly smooth shape, such as the objects of Figure 9.

3.3. Rendering

3.3.1. Algorithm

At runtime, we have to efficiently render the set of possibly overlapping volumetric billboards arbitrarily placed in the 3D scene while correctly handling transparency and levels-of-detail.

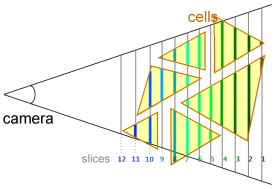


Figure 4: *Slicing: Intersection of all cells with slice 1 is drawn first, then with slice 2, and so on.*

Our solution generalizes the scheme proposed by [LDS02] dedicated to volumetric texture rendering. Instead of rendering each cell seperately, their algorithm renders them all at once (Figure 4). It generates uniformly-spaced slice planes parallel to the screen from back to front with respect to the camera. For each slice it computes all the polygons that correspond to the intersection of the given slice with all the cells. It draws these polygons and then moves to the next slice. Thus, the polygons are drawn in a back-to-front order with respect to the camera and do not need any additionnal sorting for correct transparency compositing. They are composited using the back-to-front alpha blending equation $C' = A_s\, C_s + (1 - A_s)\, C$ where C and C' are the current and new pixel colors, and A_s and C_s the alpha and color of the polygon texture.

In [LDS02] there is no overlap (their cells tile a surface). For overlapping cells, the intersection polygons belonging to one slice may overlap. This may lead to popping due to sudden change of drawing order (*e.g.*, when turning around). To avoid popping we enforce the drawing order of the polygons on a view-independent deterministic way, based on cells ID. Other restrictions of [LDS02] are that all the cells share the same texture and that the method is not aimed at complex objects or scenes: Every cell is considered for intersection with a slice, and the whole scene must be sliced (with regular sampling). Sampling empty space and intersecting all cells is especially inefficient for complex objects or scenes. We deal with these issues in the paragraph *Slab partitioning*. Moreover, at this scale perspective counts, so LOD representation and adaptive sampling should be used. Our representation accounts for LOD and our rendering does slice the space adaptively (Figure 2-*right*). To that purpose, hardware volume MIP-mapping is enabled and the distance between two successive slices is adjusted to fit the size of one voxel at its MIP-map level corresponding to the slices position, as discussed in paragraph *Adaptive slicing*.

Slab partitioning: Our algorithm avoids slicing the regions with no cells and avoids considering all the cells by partitioning the space with planes orthogonal to the viewing direction axis before drawing the cells. We define *slab* as the region between two such successive planes. The cells that intersect a slab are assigned to the slab. The slicing is then performed one slab after the other, starting from the farthest. The slabs with no cell are not sliced. For the other slabs, only the cells assigned to the slab are sliced. Within a slab we group the cells by texture, which reduces the number of texture switches required to handle cells assigned with different volumes. To ensure an effective partitioning, the slab thickness is chosen such as the slab contains at least a predefined number of slices while not being smaller than the average cells size.

Adaptive slicing: In order to avoid over or under-sampling artifacts during the slice-based volume reconstruction, we adjust the distance d between two slices to match the size r of a voxel (measured in world space). For MIP-mapped volume, r changes with the distance z of the voxel to the camera. Therefore, we also extend the slicing algorithm to account for this adaptive slicing rate.

The function used by the hardware to select the texture MIP-map level is rather complex and partly undocumented. However, our slices are parallel to the screen, which allows for the following approximation. When minification is involved, the MIP-map level is chosen such that the size of a projected voxel on screen is roughly one pixel (see Figure 5). This implies $r \approx z\frac{\tan(fov/2)}{h/2}$ where h is the height of the screen (in pixels) and fov is the camera field-of-view angle in the height direction. Thus, our algorithm adjusts d to be equal to r.

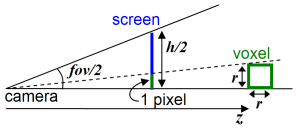


Figure 5: *Slicing rate*

To avoid over-sampling if the volume is magnified (*i.e.*, the most detailled MIP-map level has been reached and the size of a projected voxel on screen is more than one pixel), d is thresholded to be at least the magnified voxel size. This also prevents the generation of an excessive number of slices when the camera approaches or goes into a volume since the number of slices cannot exceed the volume resolution in the viewing direction.

3.3.2. Efficient prism/plane intersection

The performance of the slicing algorithm relies on the computation of the prism/plane intersection. Our aim is to efficiently compute such intersections. We design the algorithm to take advantage of GPU geometry shader.

The result of the intersection of a prism with a plane may be empty (if the plane does not intersect the prism) or may be a polygon having 3, 4 or 5 edges depending on the plane/prism configurations (Figure 6-*left*). We denote *vertical lines* as the three lines supporting the three edges of a prism that link its two base triangles (lines (0,3) (1,4) (2,5) of Figure 6-*left*). A plane may intersect the vertical lines either above, in-between or below the prism, which results in 27 possible combinations. A direct implementation of those 27 cases would be very inefficient. In order to reduce the number of tests, we analyse the problem from the point of view of each vertical line.

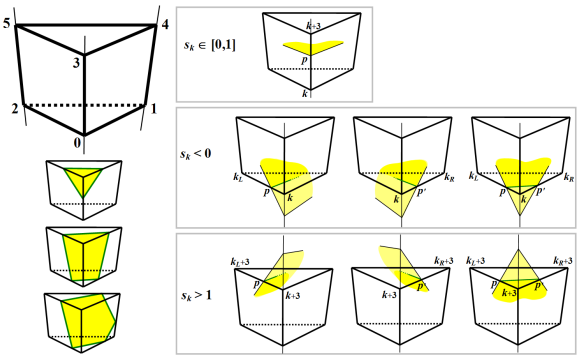


Figure 6: *Prism/plane intersection.* Left: *Notations and possible resulting polygons (0, 3, 4 or 5 edges).* Right: *The different configurations.*

The calculation is done in camera space. The algorithm is illustrated by Figure 6-*right*, and the geometry-shader implementation is outlined in Appendix A. The vertices of the prism are numbered from 0 to 5 as shown on the Figure 6-*left*. Lets consider a vertical line passing through the vertices k and $k+3$ ($k = 0$, 1 or 2). Since the slicing plane is normal to the z-axis in camera space, the plane/line intersection is defined by the factor $s_k = \frac{z_{slice}-z_k}{z_{k+3}-z_k}$, where z_{slice} is the position of the slicing plane along the z-axis. First, we reject the trivial configurations where no intersection occurs (*i.e.*, the s_k are all negative or all greater than 1). Then we perform the following operation for each vertical line successively:
- If s_k is within [0,1], one intersection point is lying between the vertices k and $k+3$. This point is computed and stored.
- If s_k is less than 0, since the no-intersection case has already been considered, the plane intersects either the edge of the prism on the left of the vertex k (the edge $[k,k_L]$) or the one on the right of k (the edge $[k,k_R]$) or both of them. This is determined by testing if the values s_{k_L} and/or s_{k_R} are positive. The one or two corresponding intersection points are then computed and stored.
- Similarly to the previous case, if s_k is greater than 1, one or two intersection points are determined by considering the edges on the left and on the right of the vertex $k+3$.

The order of the tests on s_k guarantees that an intersection point is not found twice and is found in the right order to form a valid polygon. Thus, we obtain up to 5 ordered intersection points. In the case of a geometry shader implementation, we use these points to create and emit up to 3 triangles which form the polygon resulting from the intersection of the prism with the slicing plane.

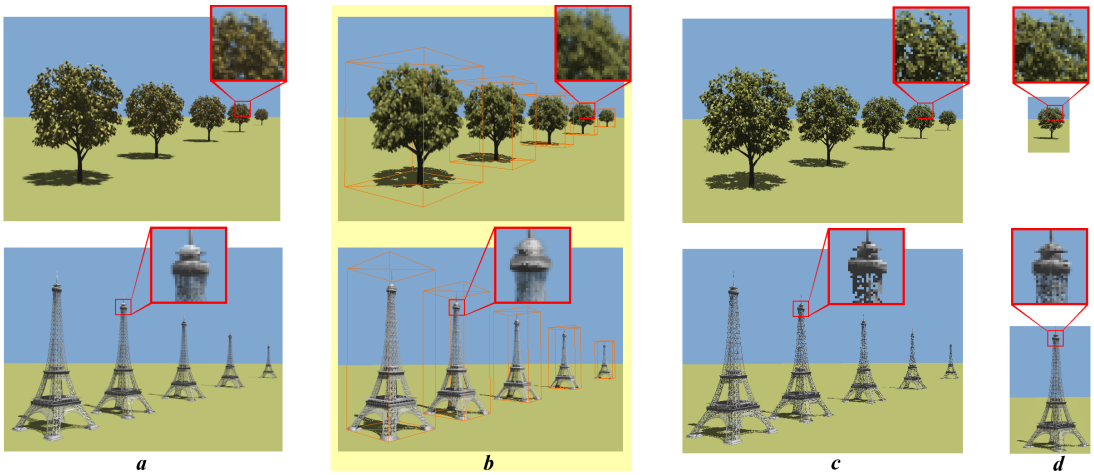


Figure 7: *Comparison. Column* ***a****: Reference images (polygonal models rendered by 3DSMax).* ***b****: Our method (*256^3 *and* $128 \times 128 \times 256$ *MIP-mapped volumes).* ***c****: Real-time rendering of the polygonal models (200000 and 22600 faces).* ***d****: Same as c with FSAA×8 enabled.*

Compared to [LDS02], our algorithm can be expressed in a compact way. It also performs less *if*-tests: In the worst case, where the intersection results in a pentagon, the number of tests is at most 12. See 4-*Performance* for benchs.

4. Results

We applied our representation to several classes of objects depicted in the figures and the accompanying video available at `http://www.antisphere.com/Research/VolumetricBillboards.php` and commented below.

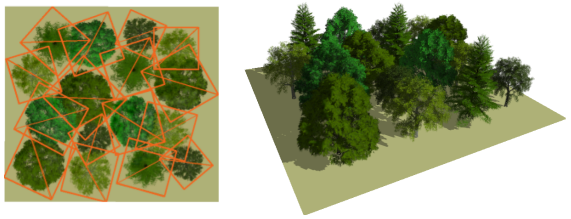


Figure 8: *A group of different trees. Note on the top-view that the cells do intersect.*

MIP-map and LOD: Figure 7-b shows the rendering quality provided by our method through the MIP-mapped volumes for levels-of-detail management. This is illustrated on two complex objects: a tree originally composed of 200 000 faces and voxelized at a resolution of 256^3; and an Eiffel tower model originally composed of 22 600 faces and voxelized at a resolution of $128 \times 128 \times 256$. We provide some additional images for quality comparison: Figure 7-a shows reference images produced from the polygonal models by an offline renderer; and Figures 7-c and 7-d show screen captures of the hardware rendering of the polygonal models without and with Full Scene Anti-Aliasing (FSAA). Note that, in our results, the LOD transitions are smooth (see video) and the objects are nicely antialiased without enabling FSAA. In contrast, the direct hardware rendering of their polygonal models shows popping or aliasing artifacts, especially at a distance, because of their very thin and ragged faces; these artifacts becomes even more noticable when the objects move. The limit of our representation is reached for closer views, when the size of the object on the screen exceeds its volume resolution, causing blurriness. Although this may be acceptable for a certain range, transitioning with the polygonal model, or with a mixed model as the tree of Figure 13, could also be considered. The nice z-buffer compliance property of the representation would ease the fading transition between the different models.

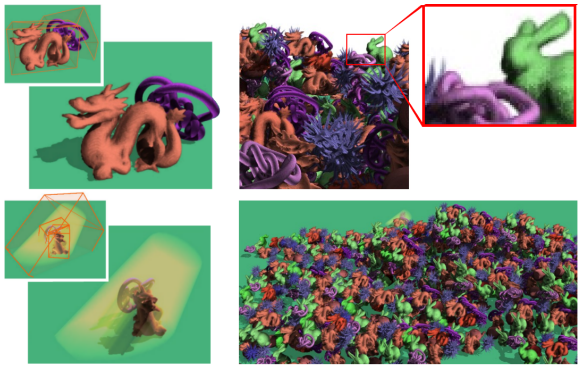


Figure 9: Left: *Overlapping semi-transparent volumes.* Right: *A scene composed of 2500 volumetric billboards displayed at interactive framerates. Note how the objects are antialiased (w/o FSAA).*

Multiple and overlapping objects: Examples depicted in Figures 8, 9 and 10 demonstrate how multiple and possibly intersecting complex objects are handled. The trees are voxelized at 128^3 resolution. The "dragons'n co" scene is composed of 2500 volumetric billboards sharing 6 different volumes with resolution ranging from 64^3 to 128^3. Despite their number and their overlapping, there are no artifacts resulting from incorrect transparency handling between objects. Furthermore, the superimposition of a fully semi-transparent volumetric billboard, like the haze cone of Fig-

ure 9-*left*, with other objects can generate interesting effects such as volumetric shadowing. The "asteroid belt" scene demonstrates the ability to handle many different volumes. It is composed of 1200 volumetric billboards sharing 60 different volumes of resolution 128^3 generated procedurally (using Perlin noise). In this example, the dust clouds overlap the asteroids. See paragraph *Perfomance* below for memory usage and performance discussion.

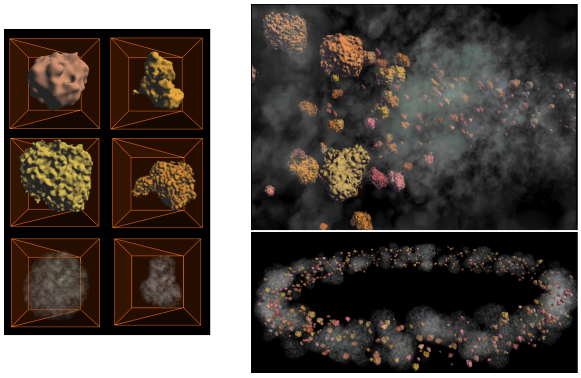


Figure 10: *"Asteroid belt" composed of 1200 volumetric billboards from 60 different volumes of resolution 128^3: 40 for asteroids, 20 for dust clouds (some examples are shown on the left).*

Volumetric textures: Volumetric billboards are also able to represent volumetric textures by generating their bounding prisms from the extrusion of a triangulated surface and its mapped texture coordinates, as shown by Figures 11 and 12. The semi-transparent MIP-mapped volumetric texture effect is obtained with no modification to the algorithm. In this context, our approach can be seen as an extension of [LDS02]. The fur and buildings volume resolutions are $256 \times 256 \times 32$.

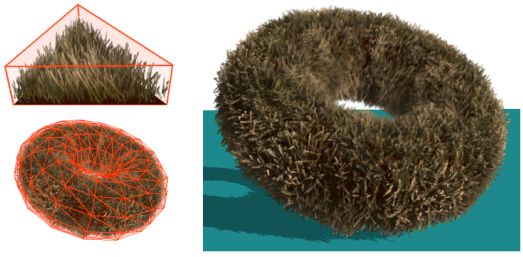


Figure 11: *Fur volumetric texture (380 cells).*

Composite and animated scene: The example of Figure 13 illustrates the ability of our representation to manage composite objects, like a tree composed of polygonal trunk and branches with leaves composed of several volumetric billboards sharing the same leaves pattern. Although the resolution of single volumetric billboards is small (128^3 in that case), compositing them results in a high-resolution model valid even for close view with full parallax and no popping occurs despite the dense overlapping. Furthermore, the

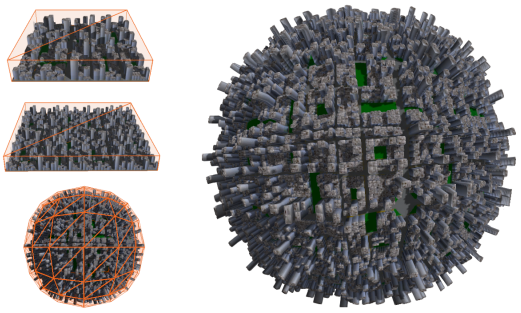


Figure 12: *Buildings block texture repeated inside a single volumetric billboard (top-left), and mapped on a sphere (192 cells).*

models can be animated by deforming or moving the prisms (see video). In a sense, this brings animation to volume data. Note that in this animation example, we simply decomposed the foliage into 12 box cells which we animate procedurally. In the scope of a real production, more cells would have been defined and linked to a deforming tree skeleton. See also the wave motion in the grass obtained by bending the cells edges.

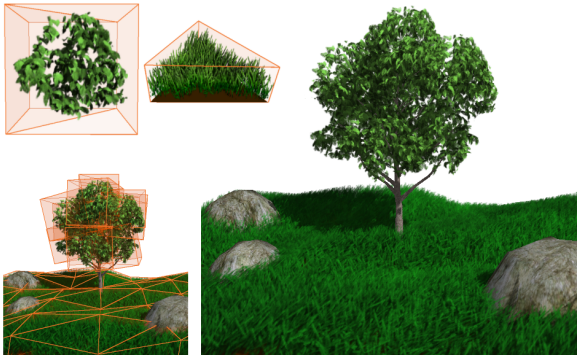


Figure 13: *Animated (see video) volumetric billboards mixed with meshes.*

Performance and memory usage: The following benchmarks (Table 1) are related to the sequences shown on the video and the figures. Framerates are measured on a Pentium 4 - 3GHz with 1GB of RAM and equiped with a GeForce 8800 GTS with 640MB of on-board memory. If nothing else specified, they correspond to a framed view (*i.e.*, the entire scene fits the screen). They are measured with shadow-mapping disabled. Performance with shadow-mapping enabled is displayed in the video.

For close views, the screen resolution has a noticable impact on the performance (except for the "asteroid belt" example), which suggests that the bottleneck is mainly the fillrate. At a distance, performance increases significantly thanks to the adaptive slicing combined with faster access to volume MIP-map levels. The “dragons’n co” scene performance is minimally affected by the screen resolution. Rather, the performance is bounded by the intersection calculations and the

		FPS (# slices)			Tex
		640×480	800×600	1024×768	(MB)
Trees (Fig. 8)	close-up	69 (230)	59 (270)	56 (320)	38
	far distance	395 (28)	336 (33)	271 (38)	
Dragons'n co (Fig. 9)	grazing view	19 (2400)	14 (2800)	11 (3300)	26
	wide view	29 (540)	25 (640)	20 (745)	
	far distance	54 (170)	51 (200)	49 (235)	
Asteroid belt (Fig. 10)	close-up	14 (978)	13 (1138)	10 (1336)	300
	wide view	9 (308)	8 (384)	6 (492)	
Furry torus (Fig. 11)		130 (300)	76 (360)	51 (410)	3
City sphere (Fig. 12)		36 (400)	24 (480)	15 (550)	3
Anim tree+grass (Fig. 13)		79 (350)	70 (415)	47 (485)	12

Table 1: *Framerates (frames per second) and average number of slices (in green between brackets) for various screen resolutions.* Last column: *Amount of compressed texture memory (in MB).*

texture switches required to render the different objects. For the asteroid belt example, the cost related to texture memory access and switches become significant, especially for low screen resolutions (at higher resolution, it is partly hidden by the fillrate cost).

Texture memory usage for each example is also reported in Table 1. The textures are compressed using the standard S3TC compression. Basically, a 256^3 volume (colors+normals) and its MIP-map levels are stored in 38 MB, and a 128^3 volume in 5 MB.

	VS slicing	GS slicing	GS + adaptive slicing	GS + adaptive slic + partitioning
Furry torus	31	68	106	130
Asteroid belt	0.6	1.3	8	14

Table 2: *Comparison.* First column*: Vertex Shader based algorithm of [LDS02].* Next columns*: Our Geometry Shader based algorithm successively enhanced by adaptive slicing and partitioning. Measured in frames per second at 640×480.*

Table 2 compares our algorithm with a GPU implementation of [LDS02] using vertex shaders (VS) on today's hardware. At the time of publication (2002) their hybrid algorithm which mixes intersection compution on GPU and CPU was reported faster than a pure GPU implementation, but it is no more the case, so we limit the comparison to GPU-based implementation. Regarding regular cell slicing only (*i.e.*, no adaptive slicing and no partitioning), despite the fact that geometry shader (GS) and dynamic branching are not yet mature fully-efficient features, our GS-based implementation outperforms the VS-based implementation by a factor of about 2. In addition, it requires 6 times less data to be sent to the GPU for geometry since the description of the cells are not duplicated at each vertex (which the VS-based algorithm requires). Regarding use in large scenes, adaptive slicing and partitioning prove to be the key component for efficiency (order of magnitude faster) and quality filtering with the distance.

Optimizations: While the results of our method are indeed interactive, there is still room for improvement. The method offers a simple way to balance between quality and performance. The adaptive slicing rate can be biased by a user defined factor to generate less slices, which leads to an undersampling of the volume. The object becomes more transparent as a side effect, however, this can be attenuated by scaling the opacity of the texture artificially (in the pixel shader) by an equivalent factor. In our examples, we used this technique to accelerate the rendering of the shadow maps.

Additionally, volumes often have large empty regions which are unnecessarily rasterized. More elaborate content-based optimization techniques may be considered for slicing-based volume rendering, and can even be used as a substitute for the early-ray termination technique of ray-casting [LMK03]. However, for volumes of relatively low resolution, as considered in our examples, the cost related to the complexification of the slicing algorithm masks the potential benefit. We found empty-region removal to be a more rewarding optimization technique (Figure 14). Instead of taking them out at runtime, the empty-spaces can be removed in a pre-processing stage by subdividing the volume and creating volumetric billboards only for the non-empty subvolumes. There are different strategies to do so; we have implemented a simple one based on a kd-tree, detailed in [VMD08]. Since the overhead due to the increase of cells is very limited and the performance is predominantly bounded by the fillrate, the gain is significant: up to 50% for the model (a) of Figure 14 and 200% for the model (b). Moreover the remaining non-empty regions may be packed to reduce the amount of required texture memory: the gain is up to 50% for the model (a) and 75% for the model (b).

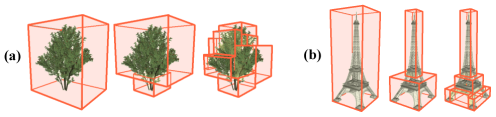


Figure 14: *Volume empty-space removal.*

5. Conclusion

The volumetric billboards method provides a solution for the representation and accelerated rendering of complex objects at a distance while not sacrificing the parallax effects. In particular, it properly handles transparency, filtering according to distance and overlapping volumes. Moreover, the real-time rendering algorithm can be integrated seamlessly into common rasterization-based renderers.

The performance of our prism/plane intersection algorithm will increase with the evolutions of geometry shaders and dynamic branching in future graphics hardwares. The limiting factor will continue to be the extensive usage of fillrate and 3D texture access. Empty-space removal adds a significant improvement in performance. In order to reduce remaining pixel overdraw, we plan to investigate front-to-back slicing schemes. Another area for future improvement is the use of a more accurate model for color,

opacity and reflectance, which takes view-dependency into account. Regarding the reflectance, correct normal filtering (*e.g.*, [HSRG07]) with a limited impact on the performance would also be of interest.

Acknowledgements

The authors would like to thank Midori Hyndman, Marie-Paule Cani, Xing Mei and Eric Bruneton for proofreading, and all members of the Evasion and Liama teams for their kind support. Philippe Decaudin is supported by a grant from the Marie-Curie project REVPE MOIF-CT-2006-22230 from the European Community.

Appendix A:

Pseudocode for prism/plane intersection geometry shader

```
ProjPosTex p[5]; // up to 5 resulting projected vert. and their tex. coord.
int i = 0;  float s0, s1, s2;
s0=(z_slice-z0)/(z3-z0); s1=(z_slice-z1)/(z4-z1); s2=(z_slice-z2)/(z5-z2);
// the slicing plane normal has been slightly jittered to avoid unwanted division by 0

if s0, s1, s2 < 0  or  s0, s1, s2 > 1 then return;

for k = 0 to 2 do  // loop to be unrolled by the compiler
    kR = (k+1) mod 3; kL = (k+2) mod 3;
    if sk ∈ [0,1] then
        p[i++] = Intersect (k, k+3)
        // Intersect(a,b) computes the intersection of the edge [a,b]
        // and the slice, and the corresponding 3D texture coordinates.
    else if sk < 0 then
        if (skL ≥ 0) then p[i++] = Intersect (k, kL)
        if (skR ≥ 0) then p[i++] = Intersect (k, kR)
    else  // sk > 1
        if (skL ≤ 1) then p[i++] = Intersect (k+3, kL+3)
        if (skR ≤ 1) then p[i++] = Intersect (k+3, kR+3)
    end
end

EmitTriangle( p[0], p[1], p[2] )
if i ≥ 4 then
    EmitTriangle( p[0], p[2], p[3] )
    if i = 5 then EmitTriangle( p[0], p[3], p[4] )
end
```

References

[ABB*07] ANDUJAR C., BOO J., BRUNET P., FAIREN M., NAVAZO I., VAZQUEZ P., VINACUA A.: Omni-directional relief impostors. *Computer Graphics Forum, Eurographics'07 26*, 3 (2007), 553–560.

[CCF94] CABRAL B., CAM N., FORAN J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *Symposium on Volume Visualization* (1994), 91–98.

[CMSW04] COOK R., MAX N., SILVA C. T., WILLIAMS P. L.: Image-space visibility ordering for cell projection volume rendering of unstructured data. *Trans. on Visualization and Computer Graphics 10*, 6 (2004), 695–707.

[CS99] CAI W., SAKAS G.: Data intermixing and multi-volume rendering. *Computer Graphics Forum 18*, 3 (1999), 359–368.

[DDSD03] DÉCORET X., DURAND F., SILLION F., DORSEY J.: Billboard clouds for extreme model simplification. *Trans. on Graphics, Siggraph'03 22*, 3 (2003), 689–696.

[DN04] DECAUDIN P., NEYRET F.: Rendering forest scenes in real-time. *Rendering Techniques'04* (2004), 93–102.

[EE02] ENGEL K., ERTL T.: Interactive high-quality volume rendering with flexible consumer graphics hardware. In *State of the art reports, Eurographics'02* (2002).

[GGSC96] GORTLER S. J., GRZESZCZUK R., SZELISKI R., COHEN M. F.: The lumigraph. In *Computer graphics and interactive techniques, Siggraph'96* (1996), pp. 43–54.

[Gri05] GRIMM S.: *Real-Time Mono- and Multi-Volume Rendering of Large Medical Datasets on Standard PC Hardware*. PhD thesis, Vienna University of Technology, 2005.

[HKRS*06] HADWIGER M., KNISS J. M., REZK-SALAMA C., WEISKOPF D., ENGEL K.: *Real-Time Volume Graphics*. A. K. Peters, 2006. ISBN: 978-1568812663.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *Symposium on Interactive 3D Graphics & Games* (2007), pp. 167–174.

[HSRG07] HAN C., SUN B., RAMAMOORTHI R., GRINSPUN E.: Frequency domain normal map filtering. *Trans. on Graphics, Siggraph'07* (2007), 28.

[JWP05] JESCHKE S., WIMMER M., PURGATHOFER W.: Image-base representations for accelerated rendering of complex scenes. In *State of the art reports Eurographics* (2005), pp. 1–20.

[KK89] KAJIYA J. T., KAY T. L.: Rendering fur with three dimensional textures. In *Computer Graphics and Interactive Techniques, Siggraph'89* (1989), pp. 271–280.

[KN01] KIM T.-Y., NEUMANN U.: Opacity shadow maps. In *Rendering Techniques'01* (2001), pp. 177–182.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *Visualization'03* (2003), pp. 38–44.

[LDS02] LENSCH H., DAUBERT K., SEIDEL H.-P.: Interactive semi-transparent volumetric textures. In *Vision, Modeling and Visualization'02* (2002), pp. 505–512.

[LH96] LEVOY M., HANRAHAN P.: Light field rendering. In *Computer graphics, Siggraph'96* (1996), pp. 31–42.

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. *Comp. graphics and interactive tech., Siggraph'94* (1994), 451–458.

[LMK03] LI W., MUELLER K., KAUFMAN A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *Visualization'03* (2003), pp. 317–324.

[MS95] MACIEL P. W. C., SHIRLEY P.: Visual navigation of large environments using textured clusters. In *Symposium on Interactive 3D Graphics* (1995), pp. 95–102.

[Nom95] NOMA T.: Bridging between surface rendering and volume rendering for multi-resolution display. In *Eurographics Workshop on Rendering* (1995), pp. 31–40.

[OBM00] OLIVEIRA M. M., BISHOP G., MCALLISTER D.: Relief texture mapping. In *Computer Graphics and Interactive Techniques, Siggraph'00* (2000), pp. 359–368.

[PO06] POLICARPO F., OLIVEIRA M. M.: Relief mapping of non-height-field surface details. In *Symposium on Interactive 3D Graphics and Games'06* (2006), pp. 55–62.

[Sch98] SCHAUFLER G.: Image-based object representation by layered impostors. In *Symposium on Virtual Reality Software and Technology'98* (1998), pp. 99–104.

[TK96] TORBORG J., KAJIYA J. T.: Talisman: commodity real-time 3D graphics for the PC. In *Computer graphics and interactive techniques, Siggraph'96* (1996), pp. 353–363.

[Tok05] TOKSVIG M.: Mipmapping normal maps. *Journal of Graphics Tools 10*, 3 (2005), 65–71.

[VMD08] VIDAL V., MEI X., DECAUDIN P.: Simple empty-space removal for interactive volume rendering. *Journal of Graphics Tools 13*, 2 (2008), 21–36.

[ZIK98] ZHUKOV S., INOES A., KRONIN G.: An ambient light illumination model. *EG Workshop on Rendering* (1998), 45–56.